BRIEF ON APPEAL

William C. Roch
Attorney for Appellant
Registration No. 24,972

SCULLY SCOTT MURPHY & PRESSER
400 Garden City Plaza
Garden City, New York 11530
(516) 742-4343

TABLE OF CONTENTS

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants:	Graham Chapman	**Examiner**:	Kenneth Tang
Serial No:	09/329,558	**Art Unit**:	2127
Filed:	June 10, 1999	**Docket**:	CA919980012US1 (12463)
For:	MAPPING A STACK IN A MACHINE ENVIRONMENT	**Confirmation**:	8251
		Dated:	September 7, 2004

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

BRIEF ON APPEAL

Sir:

This is a BRIEF ON APPEAL in support of applicant's appeal from the Examiner's decision dated December 4, 2003 finally rejecting claims 1-23. This BRIEF ON APPEAL is arranged in compliance with 37 C.F.R. §1.192(c), with subheadings and the numbers of the subheadings in conformance therewith.

Claims 1-23 are set forth in APPENDIX A attached hereto, as specified by 37 C.F.R. §1.192(c)(9).

CERTIFICATE OF MAILING UNDER 37 C.F.R. §1.8(a)

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Commissioner For Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on September 7, 2004.

Dated: September 7, 2004

William C. Roch

09/09/2004 CCHAU1 00000100 500510 09329558
01 FC:1402 330.00 DA

1. REAL PARTY IN INTEREST

The real party in interest is the assignee International Business Machines Corporation.

2. RELATED APPEALS AND INTERFERENCES

No related appeals or interferences are known to the appellant, or the appellant's legal representatives, which will directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

3. STATUS OF CLAIMS

Claims 1-23 are pending in this patent application, and are involved in this appeal.

4. STATUS OF AMENDMENTS

An AMENDMENT of April 29, 2004 was filed subsequent to the Final Rejection, and was denied entry.

5. SUMMARY OF THE INVENTION

This invention relates generally to the field of memory optimization, and provides, in particular, a method for mapping the dynamic memory stack in a programming language environment such as Java. p (page) 1, l (lines) 8-11 (line references are to line numbers in left margin)

The stack mapper of the present invention seeks to determine the shape of the stack at a given program counter. This is accomplished by locating all start points possible for a given method, that is, at all of the entry points for the method and all of the exception entry points, and trying to find a path from the beginning of the method to the program counter in question. The mapper first tries to locate a linear path from the beginning of the method, and then interactively processes the sequence of bytes at each branch until the destination program counter is reached. Once the path is found, a simulation is run of the stack through that path, which is used as the virtual stack for the purposes of the garbage collector. p 30, l 4-12

Java programs (as well as those in other object-oriented or OO languages) require the allocation of dynamic storage from the operating system at run-time. This run-time storage is allocated as two separate areas known as the "heap" and the "stack". The stack is an area of addressable or dynamic memory used during program execution for allocating current data objects and information. Thus, references to data objects and information associated with only one activation within the program are allocated to the stack for the life of the particular activation, Objects (such as classes) containing data that could be accessed over more than one activation must be heap allocated or statically stored for the duration of use during run-time. p 1, l 14-27

Because modern operating systems and hardware platforms make available increasingly large stacks, modern applications have correspondingly grown in size and complexity to take advantage of this available memory. Most applications today use a great deal of dynamic memory. Features such as multitasking and multithreading increase the demands on memory. OO programming languages use dynamic memory much more heavily than comparable serial programming languages like C, often for small, short-lived allocations. p 1, l 29 – p 2, l 4

The effective management of dynamic memory, to locate useable free blocks and to deallocate blocks no longer needed in an executing program, has become an important programing consideration. A number of interpreted OO programming languages such as Smalltalk, Java and Lisp employ an implicit form of memory management, often referred to as garbage collection, to designate memory as "free" when it is no longer needed for its current allocation. p 2, l 6-14

Serious problems can arise if garbage collection of an allocated block occurs prematurely. For example, if a garbage collection occurs during processing, there would be no reference to the start of the allocated block and the collector would move the block to the free memory list. If the processor allocates memory, the block may end up being reallocated, destroying the current processing. This could result in a system failure. p 2, l 16-23

A block of memory is implicitly available to be deallocated or returned to the list of free memory whenever there are no references to it. In a runtime environment supporting implicit memory management, a garbage collector usually scans or "walks" the dynamic memory from time to time looking for unreferenced blocks and returning them. The garbage collector starts at locations known to contain references to allocated blocks. These locations are called "roots". The garbage collector examines the roots and when it finds a reference to an allocated block, it marks the block as referenced. If the block was unmarked, it recursively examines the block for references. When all the referenced blocks have been marked, a linear scan of all allocated memory is made and unreferenced blocks are swept into the free memory list. The memory may also be compacted by copying referenced blocks to lower memory locations that were occupied by unreferenced blocks and then updating references to point to the new locations for the allocated blocks. p 2, l 25 – p 3, l 12

The assumption that the garbage collector makes when attempting to scavenge or collect garbage is that all stacks are part of the root set of the walk. Thus, the stacks have to be fully described and walkable. p 3, l 14-17

In programming environments like Smalltalk, where there are no type declarations, this is not particularly a problem. Only two different types of items, stack frames and objects, can be added to the stack. The garbage collector can easily distinguish between them and trace references relating to the objects. p 3, l 19-24

However, the Java programming language also permits base types (i.e., integers) to be added to the stack. This greatly complicates matters because a stack walker has to be more aware how to view each stack slot. Base types slots must not be viewed as pointers (references), and must not be followed during a walk. p 3, l 26-31

Further, the content of the stack may not be static, even during a single allocation. As a method runs, the stack is used as a temporary "scratch" space, and an integer might be pushed onto the stack or popped off it, or an object pushed or popped at any time. Therefore, it is important to know during the execution of a program that a particular memory location in the stack contains an integer or an object. p 4, l 1-8

The changing content of a stack slot during method execution can be illustrated with the following simple bytecode sequence of the form:

```
ICONST 0
POP
NEW
POP
RETURN p 4, l 10-18
```

As this is run, an integer, zero (0), is pushed onto the top of the stack, then popped so that the stack is empty. Then an object (pointer) is pushed onto the top of the stack, and then popped so that the stack is again empty. Schematically, the stack sequence is:

0

–

OBJECT

– p 4, l 20 – p 5, l 4

In this sequence, the constant 0 and the object share the same stack location as the program is running. Realistically, this sequence would never result in a garbage collection. However, in the naive case, if garbage collection did occur just after the integer was pushed onto the stack, the slot should be ignored, not walked, because it contains only an integer, whereas if a garbage collection occurred after the object had been pushed onto the stack, then the slot would have to be walked because it could contain the only reference to the object in the system. In addition, if the object on the stack had been moved to another location by compaction, then its pointer would have to be updated as well. p 5, l 6-18

Thus, the stack walker has to have a scheme in place to determine which elements to walk and which to skip on the stack. p 5, l 20-22

One solution proposed by Sun Microsystems, Inc in its U.S. Patent No. 5,668,999 for "System and Method for Pre-Verification of Stack Usage in Bytecode Program Loops", is to calculate the stack shapes for all bytecodes prior to program execution, and to store as a "snapshot", the state of a virtual stack paralleling typical stack operations required during the execution of a bytecode program. The virtual stack is used to verify that the stacks do not underflow or overflow. It includes multiple, pre-set entry points, and can be used as a stack map in operations such as implicit memory management. p 5, l 24 – p 6, l 3

However, the creation of a virtual stack of the whole program can be costly in terms of processing time and memory allocation, when all that may be required is a stack mapping up to a specific program counter (PC) in the stack, for a garbage collector to operate a limited number of times during program execution. p 6, l 5-l10

It is therefore an object of the present invention to provide mapping for any PC location on the stack. Then, if a garbage collection occurs, the shape of the stack can be determined for that part of the stack frame. p 6, l 13-16

It is also an object of the present invention to provide a method for mapping the shape of a portion of the stack for use either statically, at method compilation, or dynamically, at runtime. p 6, l 18-21

A further object of the invention is to provide memory optimizing stack mapping. p 6, l 23-24

The stack mapper of the present invention seeks to determine the shape of the stack at a given PC. This is accomplished by locating all start points possible for a given method, that is, at all of the entry points for the method and all of the exception entry points, and trying to find a path from the beginning of the method to the PC in question. Once the path is found, a simulation is run of the stack through that path, which is used as the virtual stack for the purposes of the garbage collector. Accordingly, the present invention provides a method for mapping a valid stack up to a destination program counter through mapping a path of control flow on the stack from any start point in a selected method to the destination program counter and simulating stack actions for executing bytecodes along said path. In order to map a path of control flow on the stack, bytecode sequences are processed linearly until the control flow is interrupted. As each bytecode sequence is processed, unprocessed targets from any branches in the sequence are recorded for future processing. The processing is repeatedly interactively, starting from the beginning of the method and then from each branch target until the destination program counter has been processed. Preferably a virtual stack is generated from the simulation, which is encoded and stored on either the stack or the heap. p 6, l 26- p 7, l 8

"The Java Virtual Machine Specification" details the set of operations that a Java virtual machine must perform, and the associated stack actions. Not included in the Java specification are some more stringent requirements about code flow. These are specified in the bytecode verifier (discussed in detail in Sun's U.S. Patent No. 5,668,999, referenced above). Code sequences that allow for different stack shapes at a given PC are not allowed because they are not verifiable. Sequences that cause the stack to grow without bound are a good example.

Thus, the following code is not legal:

```
x:      ICONST1
        GOTO x
```

because it creates an infinite loop and a never-ending stack. p 8, l 22 – p 9, l 5

The present invention is described in the context of a Java programming environment. It can also apply to any environment that prohibits the use of illegal stack statements in a manner similar to that provided by the Java bytecode verifier. p 9, l 7-11

The shape of the stack is determined by the control flows, the path or paths, within the method for which the stack frame was or will be constructed. Therefore, in the method of the present invention, a path from any start point of the method to a selected PC is located, and then the stack actions for the bytecodes along the path are simulated. The implementation of this method in the preferred embodiment is illustrated in more detail in the flow diagrams of Figures 1A and 1B, and discussed below. p 9, l 13-22

Figure 2 is a sample of stack layout 200 for a method, to illustrate the preferred embodiment. (In the example, "JSR" refers to a jump to a subroutine, a branch with a return, and "IF EQ 0" is a comparison of the top of the stack against zero.) A linear scanning of these PCs as they are laid out in memory, starting at the beginning of the method and walking forward to a selected destination, such as PC 7, is not appropriate. The linear scan would omit the jump at PC 2 to the subroutine at PC 6, resulting in a break in the stack model without knowledge of how to arrive at the selected PC. p 9, l 24 – p 10, l 2

Returning to Figure 1, the input to the method of the invention is the destination PC for the method and the storage area destination to which the resulting information on the stack shape will be written (block 100). When the mapping occurs at runtime, the definition of the storage destination will point to a location on the stack; when the mapping occurs at compile time, the pointer will be into an array for storage with the compiled method on the heap. The different uses of the invention for stack mapping at runtime and at compilation are discussed in greater detail below. p 10, l 4-14

Memory for three tables, a seen list, a branch map table and a to be walked list, are allocated and the tables are initialized in memory (block 102). In the preferred embodiment, the memory requirement for the tables is sized in the following manner. For the seen list, one bit is reserved for each PC. This is determined by looking at the size of the bytecode array and reserving one bit for each bytecode. Similarly, two longs are allocated for each bytecode or PC in both the to be walked list and the branch map table. The bit vector format provides a fast implementation. p 10, l 16-26

The three tables are illustrated schematically in Figure 3 for the code sequence given in Figure 2: Figure 3A shows the state of these tables at the beginning of the stack mapper's walk; Figures 3B through 3I show the varying states of these tables as the stack mapper walks this code sequence. p 10, l 28 – p 11, l 2

The seen list is used in the first pass of the stack mapper to identify bytes which have already been walked, to avoid entering an infinite loop. At the beginning of the walk, no bytes in the given sequence are identified as having been seen. The to be walked list provides a list of all known entry points to the method. At the beginning of the stack mapper's walk, the to be walked list contains the entry point to the method at byte zero (0) and every exception handler address for the selected method. The branch map is initially empty. p 11, l 4-13

Once these data structures are initialized, the first element from the to be walked list is selected (block 104) and the sequence of bytecodes is processed (block 106) in a straight line according to the following criteria or states and as illustrated in the flow diagram of Figure 1B. As each bytecode is selected for processing, it is added to the seen list (block 150). The actions taken in processing the bytecode are determined by the state that defines it:

state 0: flow unaffected (block 152), advance to next bytecode, if any (blocks 154, 156)

state 1:	branch conditional (block 158), if branch target has not yet been seen (block 160), then add it to the to be walked list (block 162), and in any event, advance to next bytecode, if any (blocks 154, 156)
state 2:	branch unconditional (block 164), if the branch target has not yet been seen (block 165), add it to the to be walked list (block 166) and end the straight walk (block 168)
state 3:	jump to subroutine (JSR) (block 170), if branch target has not yet been seen (block 160), the add it to the to be walked list (block 162), and in any event, advance to the next bytecode, if any (blocks 154, 156)
state 4:	return (block 172) ends the straight walk (block 168)
state 5:	table bytecode (block 174), if branch targets have not yet been seen (block 165), then add them to the to be walked list (block 166), and end the straight walk (block 168)
state 6:	wide bytecode (block 176), calculate size of bytecode to determine increment to next bytecode (block 178) and advance to next bytecode, if any (blocks 154, 156)

state 7: breakpoint bytecode (block 180), retrieve the actual bytecode and its state (block 182), and then process the actual bytecode (starting at block 150) p 11, l 15 – p 12, l 28

State 0 defines a byte that does not cause a branch or any control flow change. For example, in the sample sequence of Figure 2, A LOAD does not affect the flow and would be processed as state 0. p 12, l 30 – p 13, l 2

A conditional branch (state 1) has two states; it can either fall through or go to destination. As the stack mapper processes a conditional branch, it assumes a fall through state, but adds the branch target to the to be walked list in order to process both sides of the branch. Figure 2 contains a conditional branch at bytes 4, 5. However, if a branch target has already been walked (according to the seen list), then the target is not added (block 156 in Figure 1B). p 13, l 4-12

A JSR is a language construct used in languages like Java. It is similar to an unconditional branch, except that it includes a return, similar to a function call. It is treated in the same way as a conditional branch by the stack mapper. Figure 2 contains a JSR to byte 6 at byte 2. p 13, l 14-19

Table bytecodes includes lookup tables and table switches (containing multiple comparisons and multiple branch targets). These are treated as an unconditional branch with multiple branches or targets; any targets not previously seen according to the seen list are added to the to be seen list. p 13, l 21-26

Temporary fetch and store instructions are normally one or two bytes long.

One byte is for the bytecode and one byte is for the parameter unless it is inferred by the bytecode. However, Java includes an escape sequence which sets the parameters for the following bytecode as larger than normal (wide bytecode). This affects the stack mapper only in how much the walk count is incremented for the next byte. It does not affect control. p 13, l 28 – p 14, l 4

Breakpoints are used for debugging purposes. The breakpoint has overlaid the actual bytecode in the sequence, so is replaced again by the actual bytecode. Processing of the bytecodes in the sequence continues until terminated (eg., by an unconditional branch or a return), or when there are no more bytecodes in the sequence. Returning to Figure 1A, if the selected PC was not seen during the walk because it is not found on the seen list (block 108), the next element on the to be walked list is selected (block 104) and the bytecode sequence from it processed (block 106) following the same steps in Figure 1B until the selected PC has been walked (block 108 in Figure 1A). p 14, l 6-18

Thus, the processing of the bytecode sequence in Figure 2, given PC7 as the destination PC, would be performed as follows:

Figure 3A: At commencement, there would be only one element, PC 0 in the to be seen list 308.

Figure 3B: PC 0 is marked as "seen" in the seen list 310 and removed from the to be seen list 312. A LOAD does not affect the control flow; it is state 0. The stack mapper moves on to the next byte, PC 1.

Figure 3C: PC 1 is marked as "seen" in the seen list 314. The byte is again A LOAD, state 0, so the stack mapper moves on to the next PC.

Figure 3D: PC 2 ("JSR") is treated in the first pass as a conditional branch. Once PC 2 is added to the seen list 316, its target PC 6 is added to the to be walked list 320 and the branch PC 6 (destination PC 304) / PC 2 (source branch 306) is added to the branch list 318.

Figure 3E: The I LOAD of PC 3 is state 0, so the stack mapper moves to the next byte after adding PC 3 to the seen list 322.

Figure 3F: PC 4 is a conditional branch. After adding PC 4 to the seen list 324, the stack mapper attempts to add its target, PC 0, to the to be seen list 320, but cannot because PC 0 is already on the seen list 324.

Figure 3G: At the return of PC 5, code flow stops (state 4), ending the stack walk after PC 5 has been added to the seen list 326. p 14, l 20 – p 15, l 27

At this point, the stack mapper determines whether it has seen the destination PC 7 (as per block 108 in Figure 1A). Since it has not, the stack mapper begins processing a new line of bytecodes from the next entry on the to be walked list (block 104). According to the sample of Figure 2, the next PC on the to be walked list 320 in Figure 3G) is PC 6, the conditional branch from PC 2. Therefore, after marking PC 6 as seen (seen list 330, Figure 3H), the stack mapper processed the PC according to state 0 and proceeds to the next bytecode, which is PC 7. PC 7 is marked as seen (seen list 332, Figure 3I), and the walk ends again because it has encountered a fresh return (state 4). p 15, l 29 – p 16, l 9

Once the selected PC has been walked (block 108), the path to the destination is calculated in reverse (block 110) by tracing from the destination PC 304 to the source PC 306 on the branch map list. In the example, the reverse flow is from PC 7 to PC 6 to PC 2. Because there is no comparable pairing of PC 2 with any other designated PC, it is assumed that PC 2 flows, in reverse, to PC 0. The reverse of this mapping provides the code flow from the beginning of the method to the destination PC 7, that is:

PC 0 -> PC 2 -> PC 6 -> PC 7.

This is the end of the first pass of the stack mapper over the bytecodes. p 16, l 11-26

In the second pass, the stack mapper creates a simulation of the bytecodes (block 112) during which the stack mapper walks the path through the method determined from the first pass simulating what stack action(s) the virtual machine would perform for each object in this bytecode sequence. For many of the bytecode types (eg., A LOAD), the actions are table driven according to previously calculated stack action (pushes and pops) sequences. p 16, l 27 – p 17, l 3

Fifteen types of bytecodes are handled specially, mainly because instances of the same type may result in different stack action sequences (eg., different INVOKES may result in quite different work on the stack). p 17, l 5-8

An appropriate table, listing the table-driven actions and the escape sequences in provided in the Appendix hereto. A virtual stack showing the stack shape up to the selected PC is constructed in memory previously allocated (block 114). In the preferred embodiment, one CPU word is used for each stack element. The virtual stack is then recorded in a compressed encoded format that is readable by the virtual machine (block 116). In the preferred embodiment, each slot is compressed to a single bit that essentially distinguishes (for the use of the garbage collector) between objects and non-objects (eg., integers). P 17, l 10-21

The compressed encoded stack map is stored statically in the compiled method or on the stack during dynamic mapping. In the case of static mapping, a stack map is generated and stored as the method is compiled on the heap. A typical compiled method shape for a Java method is illustrated schematically in Figure 4. The compiled method is made up of a number of fields, each four bytes in length, including the object header 400, bytecodes 402, start PC 404, class pointers 406, selector 408, Java flags 410 and literals 414. According to the invention, the compiled method also includes a field for the stack map 412. The stack map field 412 includes an array that encodes the information about the temps or local variables in the method generated by the stack mapper in the manner described above, and a linear stack map list that a garbage collector can use to access the stack shape for a given destination PC in the array by calculating the offset and locating the mapping bits in memory. p 17, l 23 – p 18, l 9

A stack map would normally be generated for static storage in the compiled method when the method includes an action that transfer control from that method, such as invokes, message sends, allocates and resolves. p 18, l 11-14

The stack map can also be generated dynamically, for example, when an asynchronous event coincides with a garbage collection. To accommodate the map, in the preferred embodiment of the invention, empty storage is left on the stack. p 18, l 16-20

Figure 5 illustrates a stack frame 500, having standard elements, such as an area for temps or arguments pushed by the method 502, literals or the pointer the compiled method 504 (which also gives access to the stack map in the compiled method) and a back pointer 506 pointing to the previous stack frame. A small area of memory 508, possibly only four bytes, is left empty in the frame but tagged as needing dynamic mapping. An advantage of this is that if this stack frame 500 is deep in the stack, once the dynamic mapping has taken place, the frame will be undisturbed and is available for future activations. The area on the stack for dynamic stack mapping 508 can be allocated whenever a special event occurs such as timer or asynchronous events and debugging, as well as for invokes, allocates and resolves discussed above. p 18, l 22 – p 19, l 4

6. <u>CONCISE STATEMENT OF THE ISSUES PRESENTED FOR REVIEW</u>

Whether claims 1-3, 6-7, 9-13, 16-17, and 19-23 are unpatentable under 35 U.S.C. §103(a) over Agesen (U.S. 6,047,125) in view of Gosling (U.S. 5,668,999).

Whether claims 4-5, 14-15 are unpatentable under 35 U.S.C. §103(a) over Agesen (U.S. 6,047,125) in view of Agesen (U.S. 5,909,579).

Whether claims 8 and 18 are unpatentable under 35 U.S.C. §103(a) over Agesen (U.S. 6,047,125) in view of Gosling (U.S. 5,668,999) and further in view of O'Connor (U.S. 6,098,089).

7. GROUPING OF CLAIMS

The claims of each group do not stand or fall together. Claims 1, 11 and 23 are basic independent claims describing the present invention. However, each of dependent claims 2-19 and 12-19 is believed to present separate issues of patentability with respect to the prior art, and the patentability of these claims is argued separately in Section 8 of this BRIEF ON APPEAL.

8. APPELLANT'S ARGUMENTS WITH RESPECT TO EACH OF THE ISSUES ON APPEAL

The present invention concerns a method for mapping a valid stack up to a destination program counter, wherein the method maps a path of control flow on the stack from any start point in a selected method to the destination program counter and simulating stack actions for executing said existing bytecodes along said path.

In the present invention, a *single* path is mapped from a start point to the destination program counter. This aspect can be particularly seen in Figure 1A, wherein the search for a destination program counter (PC) is shown as a one-cycle pass. Figure 1A shows that if a destination PC is not seen at step 108, then the next PC is selected from a list, and conversely, if a destination PC is seen at step 108, then a reverse map of the path is created. Therefore, one can assess that the concept of performing a single mapping pass through the code and stopping the mapping as soon as the first path is established as presently claimed is inherently and explicitly expressed in the present application. *See*, Specification, page 6, lines 5-16, Figure 1A.

Agesen I

In regard to the Examiner's rejection of Claims 1-3, 6, 7, 9-13, 16, 17 and 19-23 under 35 U.S.C. §103(a) as being unpatentable over Agesen U.S. 6,047,125 (hereafter Agesen I) in view of Gosling, applicants respectfully disagree for the following reasons.

Particularly, in distinction to Agesen I, whether taken alone or in combination with Gosling, the present invention is directed to a method for mapping a stack in a stack machine environment to help determine the shape of the stack at a given program counter. As now set forth in Claims 1, 11 and 23, this is accomplished by locating all start points possible for a given method, that is, at all of the entry points for the method and all of the exception entry points, and trying to find a path from the beginning of the method to the program counter in question, and then iteratively processing the sequence of bytes at each branch until the destination program counter is reached. Once the path is found, a simulation is run of the stack through that path, which is used as the virtual stack for the purposes of the garbage collector.

Unique to the present invention is the concept that the path mapping stops, i.e. the path is complete, when the destination PC is reached for the first time. That is, there may be more than one path to get from the beginning of the method to the destination PC, but according to the invention, the mapping may stop as soon as the first path is established.

Independent Claims 1, 11 and 23 set forth a method step for mapping a valid stack up to a destination program counter including mapping a path of control flow on the stack from any start point in a selected method to the destination program counter by locating a linear path from the beginning of the method to the destination program counter and iteratively processing an existing bytecode sequence at each branch, and identifying the path as complete when the destination counter is reached. Therefore, the objective of the present invention, unlike Agesen I whose objective is to modify bytecodes, is to simplify and make more economical the preparation for garbage collection that is beneficial and neither addressed nor suggested in the cited prior art.

Thus it happens that virtually the only reason to perform this with current and probably future Java machines is to facilitate garbage collections (*See* discussion in the first full paragraph of the prior art section of Page 6 of the present application). If there were other reasons to produce a stack map, however, the invention would also be applicable.

Further, in regard to the step of identifying the path as complete when the destination counter is reached, page 7, lines 13-16 set forth "The processing is repeated iteratively, starting from the beginning of the method and then from each branch target until the destination program counter has been processed." Further, in Figure 1A, decision block 108 determines whether the destination PC has been seen, and if so, the path has been completed and the system creates at block 110 the reverse map of the path. <u>This means that when even one path to the destination PC has been walked, the method of the invention will stop the path-mapping phase of the process</u>.

Respectfully, none of the cited prior art documents discloses analyzing only enough bytecodes to establish a path from the beginning of a program or method, as clearly provided in the present claims. Conversely, Agesen I describes walking through the entire sequence of instructions, not stopping until every instruction has been analyzed for conflicts (col.

5, lines 30-37, col. 5, lines 40-43, col. 9, lines 54-57, col. 9, line 66 to col. 10, line 2 and Figure 5, blocks 510, 530).

Gosling

Similarly, Gosling discloses processing all the instructions in a program (col. 7, lines 33-35 and Figure 4A, block 410).

Agesen II, Agesen I

Initially, the rejection of claims 4-5 and 14-15 over Agesen I in view of Agesen II seems a little odd since claims 4-5 and 14-15 depend from respectively from claims 2-3 and 12-13 which stand rejected over a different combination of prior art.

Agesen U.S. 5,909,579 (hereafter Agesen II) discloses analyzing each bytecode (col. 7, lines 47-52) and storing the live pointer information for each bytecode (col. 7, lines 35-37, col. 9, lines 24-28 and Figure 4, block 408). Thus, none of the cited references whether taken alone or in combination, teaches or suggests the ability for terminating the path-mapping phase of the process upon reaching the destination counter.

Agesen I is essentially directed to a method for modifying a sequence of instructions to improve memory management within a storage device during execution of the instructions, which comprises steps, performed by a processor, of: (a) analyzing the sequence of instructions for a conflict indicating an undeterminable variable type, (b) determining the type of conflict and (c) modifying the sequence of instructions to eliminate the conflict based on the determination.

Further, as stated in the Summary (Col. 5, lines 16 et seq. of Agesen I patent) the subject matter of Agesen is to improve garbage collection by modifying code that introduces a conflict in the assignment of variables for identifying references. The conflict exists, for example, when a method includes code defining at least two control paths leading to a common subroutine and the garbage collector initiated during execution of the subroutine cannot determine whether a variable represents a reference. By rewriting the code in Agesen I the conflicts are eliminated.

With respect to the Examiner's rejection of Claims 4-5 and 14-15 under 35 U.S.C. §103(a) as allegedly being unpatentable over Agesen I in view of Agesen II, Agesen II describes, as stated in the summary of the invention at col. 3, line 63 to col.4, line 17, a method in which:

> [L]ive pointer information for a stream of bytecodes is precomputed for each bytecode. The precomputed full live pointer information is stored only for bytecodes at predetermined intervals in the stream. Between the bytecodes for which full live pointer information is stored, changes in the live pointer information produced by each bytecode are encoded using a suitable compressive coding and stored. Later, when garbage collection is initiated, the full live pointer information for the nearest bytecode preceding the desired bytecode boundary is retrieved along with the intervening coded changes. The changes are decoded and applied to the retrieved live pointer information to generate the live pointer information at the desired bytecode boundary. In accordance with one embodiment of the invention, the live pointer changes are delta encoded so that each code contains information relating to the live pointer changes produced by a bytecode from the live pointer information as modified by the previous delta code. In accordance with another embodiment of the invention, the delta coded changes are encoded with a Huffman encoding scheme.

The basic invention in Agesen II is choosing to store full information for only every *n*th bytecode in the stream, and recreating the deltas in between them. The Examiner states that Agesen II "teaches a 'bytecode analyzer mechanism' which determines the changes which the bytecode makes to the live pointer locations (col.8, lines 20-24)." He further states that "the system has breakpoints or computation stops at bytecode boundaries for determining live pointer information (col.3, lines 10-13)."

The applicants respectfully submit that Agesen I and Agesen II combined do not disclose either mapping a path of control flow or simulating stack actions including iteratively processing an existing bytecode sequence at each branch, and identifying the path as complete when the destination counter is reached, as discussed above with respect to Claims 1 and 11, from which Claims 4, 5, 14 and 15 depend. Neither do they disclose any steps that are similar to, or that suggest the use of, those two elements. Thus the two references do not contain or suggest all of the elements of the claims.

With respect to the rejection of Claims 6, 7, 9, 16, 17 and 19 under 35 U.S.C. §103(a) as allegedly being unpatentable over Agesen I in view of Gosling, the Examiner states that Gosling supplies, col. 5, lines 21-29, the step of generating a virtual stack that is omitted from Agesen I. However, Claims 1 and 11, from which Claims 6 and 16 respectively depend, are restricted to mapping a path of control flow only from an arbitrary start point up to a destination program counter, and identifying the path as complete when the destination counter is reached. Therefore the virtual stack thus created is only a stack for part of the program, unlike Gosling that creates a virtual stack for the entire program, i.e., "used in the same way as a regular stack," (*See* col.5, lines 35-40 of Gosling). That is, the method of the present invention eliminates the time and cost of generating a complete virtual stack in the manner of Gosling.

In addition to the foregoing analysis and arguments, the following comments represent arguments received from the inventors Ray Saunders and Trent Gray – Donald on the proposed combinations of prior art applied in the Final Rejection.

In general, the present invention, as noted at page 7, lines 3-16, is directed to

> "...a method for mapping a valid stack up to a destination program counter through mapping a path of control flow on the stack from any start point in a selected method to the destination program counter and simulating stack actions for executing bytecodes along said path. In order to map a path of control flow on the stack, bytecode sequences are processed linearly until the control flow is interrupted. As each bytecode sequence is processed, unprocessed targets from any branches in the sequence are recorded for future processing. The processing is repeatedly iteratively, starting from the beginning of the method and then from each branch target until the destination program counter has been processed."

<u>Agesen I (or 1)</u>

According to the summary of the invention, at col.5, 11.29-36, Agesenl is directed to:

> "a method for modifying a sequence of instructions to improve memory management within a storage device during execution of the instructions, comprises the steps, performed by a processor, of (a) analyzing the sequence of instructions for a conflict indicating an undeterminable variable type, (b) determining the type of conflict, and (c) modifying the sequence of instructions to eliminate the conflict based on the determination."

The substance of the invention in Agesenl is that it provides for modifying the program bytecodes in the stack, col.5, lines 23-25. Every embodiment of Agesenl includes the step of rewriting some code in the program.

The present invention does not define in any manner rewriting any code in the program. Further, it deals with the dynamic nature of the stack by leaving open a small area of memory in the stack and tagging it for dynamic mapping. Agesenl does not describe a similar approach at all.

The examiner states that Agesenl teaches "simulating stack actions for executing bytecodes," col.3, lines 62-65 and col. 11, lines 59-63. On a careful reading of Agesenl, there is nothing approaching simulating stack actions at the cited passage or any other passage. Agesen 1 deals with instructions in code being processed, as exemplified in the cited passage at col.5, 1.32, "during execution of the instructions".

The passage at col.3, 11.62-65 is not a statement of any operation performed by Agesenl, but merely an example of a bytecode (i add) that is found in the target program.

As to "execution" at col. 11, lines 59-63, the code in Figure 7 is also an example of what bytecodes the Agesen 1 invention might encounter in code created by a Java compiler - see the general description of Figure 7 at col. 10, lines 49-52. Further, described at col. 11, lines 64-65, are the conflicts that the Agesenl invention is intended to resolve.

The examiner states that Agesenl describes "mapping a path of control flow on the stack from any start point in a selected method to the destination program counter by locating a linear path from the beginning of the method to the destination program counter and iteratively processing a bytecode sequence for each branch until said destination program counter is reached."

In support of this contention, the examiner states that at col.7, 11.1-5, Agesenl uses a stack map to map controls to a program counter. The sentence immediately preceding the cited passage is critical to the passage, the whole of which reads: "The conflicts are caused by certain conditions set by rules and exceptions dictated by Java's bytecode verifier. They made it impossible for a collector, which relies on only the program counter to determine the stack map to use, to determine conclusively whether a local variable with an assigned stack frame slot represents a reference. Such conflicts are eliminated by rewriting selected bytecodes to refer to a different variable or adding additional bytecodes to initialize the variable." The applicant submits that the complete passage describes a condition in the Java code that the invention of Agesen 1 is intended to resolve. Neither the cited passage nor any other passage in Agesenl describes locating a linear path from the beginning of a method to a destination program counter to map a path of control flow.

Therefore the applicant respectfully submits that Agesenl discloses neither of the steps in any of claims 1-3, 10-13 and 20-23 nor any similar step.

The Examiner has stated that the features not shown in the reference, and relied upon by the applicant, are not recited in the claims. The applicant submits that the particular features are discussed to show that the reference is directed to resolving conflicts indicating an undeterminable variable type in garbage collection, and is not relevant to the present invention which is directed to mapping the shape of the stack. The present invention is not a garbage collection method, although it may be used prior to garbage collection.

The examiner, in the rejection of claims 2, 3, 12 and 13, states that Agesenl teaches processing a first linear bytecode sequence and an additional one until the control flow is interrupted. The reference to col.5, lines 19-27, shows an example of a conflict that Agesenl attempts to resolve, and not how Agesenl resolves it. As to the cited passage at col.7, 11.42-44, the interrupt controller, 135 in Figure 3, is merely a component of a standard well-known computer system. It is not shown in Agesenl that the interrupt controller performs a particular step similar to steps in claims 2, 3, 12 or 13.

As to the passage at col. 11, lines 54-59, cited by the examiner, the applicant submits that while an exception handler as described by Agesenl may in some situations interrupt control flow, Agesenl does not show that any unprocessed targets are recorded for future processing. The only item recorded is the reference for the exception that caused entry into the exception handler, which is quite different from the recording step of the rejected claims.

The examiner states that "Ageson inherently teaches: the destination program counter was not reached during an earlier processing of a linear bytecode sequence [and] the information stored in the reference can be used at a future time for processing." The information stored in the reference is not information about an unprocessed target. Rather it is about an exception that caused entry into the handler. Applicant submits that the cited passage does not disclose a step similar to the steps of claims 2, 3, 12 or 13, and that the rejection based thereon is in error.

Agesen 2

Agesen2 describes, as stated in the summary of the invention at col.3, line 63 to col.4, line 17, a method in which:

> "...live pointer information for a stream of bytecodes is precomputed for each bytecode. The precomputed full live pointer information is stored only for bytecodes at predetermined intervals in the stream. Between the bytecodes for which full live pointer information is stored, changes in the live pointer information produced by each bytecode are encoded using a suitable compressive coding and stored. Later, when garbage collection is initiated, the full live pointer information for the nearest bytecode preceding the desired bytecode boundary is retrieved along with the intervening coded changes. The changes are decoded and applied to the retrieved live pointer information to generate the live pointer information at the desired bytecode boundary. In accordance with one embodiment of the invention, the live pointer changes are delta encoded so that each code contains information relating to the live pointer changes produced by a bytecode from the live pointer information as modified by the previous delta code. In accordance with another embodiment of the invention, the delta coded changes are encoded with a Huffman encoding scheme."

The basic invention in Agesen2 is choosing to store full information for only every nth bytecode in the stream, and recreating the deltas in between them.

The examiner states that Agesen2 "teaches a 'bytecode analyzer mechanism' which determines the changes which the bytecode makes to the live pointer locations (col.8, lines 20-24)." He further states that "the system has breakpoints or computation stops at bytecode boundaries for determining live pointer information (col.3, lines 10-13)."

The applicant submits that Agesenl and Agesen2 combined do not disclose either mapping a path of control flow or simulating stack actions. Neither do they disclose any steps that are similar to, or that suggest the use of, those two elements. Thus since the two references do not contain or suggest all the elements of the claims, the applicant submits respectfully that the invention of claims 4, 5, 14 and 15 is not obvious over the combined references.

Gosling

Re claims 6 and 16, the examiner states that Gosling supplies, col.5, lines 21-29, the step of generating a virtual stack that is missing from Agesenl. However, claims 1 and 11, from which claims 6 and 16 respectively depend, are restricted to mapping a path of control flow only from an arbitrary start point up to a destination program counter. Therefore the virtual stack thus created is only a stack for part of the program, unlike Gosling which creates a virtual stack for the entire program "used in the same way as a regular stack", col.5, lines 35-40.

Thus the combined references of Agesenl and Gosling do not include any similar steps that would suggest all the steps of claims 6 and 16, which depend from claims 1 and 11 respectively.

Re claims 9 and 19, depending from claims 7 and 17 respectively, the storing of a bitstring to a pre-allocated area on the stack is described at page 19, lines 1-4 of the disclosure. Although storing information to a pre-allocated area is known in general, the claimed pre-allocated area is used when a special event occurs. In contrast, Agesenl stores the value at a position selected not by the invention but by the compiler: "The compiler selected variable 3 to store the value for itmp so bytecode 704 stores the value (1) from the top of the operand stack in variable 3." This storage is simply matching the storage location to the location chosen by the compiler.

O'Connor

Re claims 8 and 18, the examiner states that although neither Agesenl nor Gosling teaches storing the bitstring on a heap, O'Connor states that it is well known in the art that "garbage collection" of "heap-allocated storage" is an "attractive model for dynamic memory management", col. 1, lines 39-42.

Claims 8 and 18 add the limitation that the step of storing the bitstring comprises storing the bitstring to the selected method as compiled on a heap. The cited passage from O'Connor does not indicate anything about the stack map being placed in the heap within the compiled method. In general, O'Connor discusses ordinary stores to the heap but not the storage of a stack map.

9. **CONCLUSION**

In view of the above, it is respectfully submitted that the Final Rejection is in error and should be reversed for good reasons, and it is respectfully requested that the Board of Patent Appeals and Interferences so find.

Respectfully submitted,

William C Roch

William C. Roch
Registration No. 24,972

SCULLY, SCOTT, MURPHY & PRESSER
400 Garden City Plaza
Garden City, New York 11530
(516) 742-4343

WCR/jf

APPENDIX A

Claim 1 A method for mapping a valid stack up to a destination program counter, comprising:

mapping a path of control flow on the stack from any start point in a selected method to the destination program counter by locating a linear path from the beginning of the method to the destination program counter and iteratively processing an existing bytecode sequence for each branch, and identifying said path as complete when said destination program counter is reached; and

simulating stack actions for executing said existing bytecodes along said path, and constructing a virtual stack for storage in a pre-allocated memory location~~,~~

Claim 2 The method of claim 1 wherein the step of mapping a path of control flow on the stack comprises:

processing a first linear bytecode sequence until the control flow is interrupted; and recording unprocessed targets from any branches in the first linear bytecode sequence for future processing.

Claim 3 The method of claim 2 wherein the step of mapping a path of control flow on the stack further comprises:

processing an additional bytecode linear sequence until the control flow is interrupted; and

recording unprocessed targets from any branches in the additional linear bytecode sequence for future processing, where the destination program counter was not reached during an earlier processing of a linear bytecode sequence.

Claim 4 The method of claim 2 wherein the step of processing any linear bytecode sequence comprises:

determining if a bytecode in said any linear bytecode sequence is a breakpoint with a pointer to bytecode data; and

replacing the breakpoint with the bytecode data.

Claim 5 The method of claim 3 wherein the step of processing any linear bytecode sequence comprises:

determining if a bytecode in said any linear bytecode sequence is a breakpoint with a pointer to bytecode data; and

replacing the breakpoint with the bytecode data.

Claim 6 The method of claim 1 wherein the step of simulating stack actions executing the bytecodes along the path further comprises generating a virtual stack.

Claim 7 The method of claim 6, further comprising:

encoding the virtual stack as a bitstring and storing the bitstring at a selected destination for use in memory management operations.

Claim 8 The method of claim 7, wherein the step of storing the bitstring comprises storing the bitstring to the selected method as compiled on a heap.

Claim 9 The method of claim 7, wherein the step of storing the bitstring comprises storing the bitstring to a pre-allocated area on the stack.

Claim 10 The method of claim 1 wherein the step of simulating stack actions executing the bytecodes along the path further comprises:

inserting pre-determined stack actions for bytecodes maintaining the control flow in the selected method; and

calculating stack actions for bytecodes transferring the control flow from the selected method.

Claim 11 A method for mapping a Java bytecode stack up to a destination program counter comprising:

mapping a path of control flow on the stack from any start point in a selected method to the destination program counter by locating a linear path from the beginning of the method to the destination program counter and iteratively processing an existing bytecode sequence at each branch, and identifying said path as complete when said destination counter is reached; and

simulating stack actions for executing said existing bytecodes along said path, and constructing a virtual stack for storage in a pre-allocated memory location.

Claim 12 The method of claim 11 wherein the step of mapping a path of control flow on the stack comprises:

processing a first linear bytecode sequence until the control flow is interrupted; and

recording unprocessed targets from any branches in the first linear bytecode sequence for future processing.

Claim 13 The method of claim 12 wherein the step of mapping a path of control flow on the stack further comprises:

processing an additional bytecode linear sequence until the control flow is interrupted; and

recording unprocessed targets from any branches in the additional linear bytecode sequence for future processing, where the destination program counter was not reached during an earlier processing of a linear bytecode sequence.

Claim 14 The method of claim 12 wherein the step of processing any linear bytecode sequence comprises:

determining if a bytecode in said any linear bytecode sequence is a breakpoint with a pointer to bytecode data; and

replacing the breakpoint with the bytecode data.

Claim 15 The method of claim 13 wherein the step of processing any linear bytecode sequence comprises:

determining if a bytecode in said any linear bytecode sequence is a breakpoint with a pointer to bytecode data; and

replacing the breakpoint with the bytecode data.

Claim 16 The method of claim 11 wherein the step of simulating stack actions executing the bytecodes along the path further comprises generating a virtual stack.

Claim 17 The method of claim 16 further comprising:

encoding the virtual stack as a bitstring and storing the bitstring at a selected destination for use in memory management operations.

Claim 18 The method of claim 17, wherein the step of storing the bitstring comprises storing the bitstring to the selected method as compiled on a heap.

Claim 19 The method of claim 17, wherein the step of storing the bitstring comprises storing the bitstring to a pre-allocated area on the stack.

Claim 20 The method of claim 11 wherein the step of simulating stack actions executing the bytecodes along the path further comprises:

inserting pre-determined stack actions for bytecodes maintaining the control flow in the selected method; and

calculating stack actions for bytecodes transferring the control flow from the selected method.

Claim 21 A computer-readable memory for storing the instructions for use in the execution in a computer of the method of claim 1.

Claim 22 A computer readable memory for storing the instructions for use in the execution in a computer of the method of claim 11.

Claim 23 A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for mapping a valid stack up to a destination program counter, said method steps comprising:

mapping a path of control flow on the stack from any start point in a selected method to the destination program counter and identifying said path as complete when said destination counter is reached; and

simulating stack actions for executing existing bytecodes along said path, wherein the step of mapping a path of control flow on the stack comprises:

processing a first linear existing bytecode sequence until the control flow is interrupted; and

recording unprocessed targets in a pre-allocated memory location from any branches in the first linear existing bytecode sequence for future processing, and

where the destination program counter was not reached during an earlier processing of a linear existing bytecode sequence,

processing an additional existing bytecode linear sequence until the control flow is interrupted; and

recording unprocessed targets in said pre-allocated memory location from any branches in the additional linear existing bytecode sequence for future processing.